Secrecy by Typing and File-Access Control

Avik Chaudhuri Martín Abadi*
Computer Science Department
University of California, Santa Cruz

Abstract

Secrecy properties can be guaranteed through a combination of static and dynamic checks. The static checks may include the application of special type systems with notions of secrecy. The dynamic checks can be of many different kinds; in practice, the most important are access-control checks, often ones based on ACLs (access-control lists). In this paper, we explore the interplay of static and dynamic checks in the setting of a file system. For this purpose, we study a pi calculus with file-system constructs. The calculus supports both access-control checks and a form of static scoping that limits the knowledge of terms—including file names and contents—to groups of clients. We design a system with secrecy types for the calculus; using this system, we can prove secrecy properties by static typing of programs in the presence of file-system access-control checks.

1 Introduction

Secrecy properties can be guaranteed through a combination of static and dynamic checks. The static checks may include the application of special type systems with notions of secrecy (*e.g.*, [1, 2, 7]). The dynamic checks can be of many different kinds; in practice, the most important are access-control checks, often ones based on ACLs (access-control lists). In this paper, we explore the interplay of such static and dynamic checks.

The setting of our study is a fairly standard file system. This setting is attractive because storage systems often include rather structured and sometimes interesting facilities for dynamic checks. In particular, ACLs are often central to storage systems; network-attached storage systems also feature elaborate cryptographic implementations of access control (*e.g.*, [11, 15]). This setting is attractive also because of the practical importance of secure storage, and because the rigorous analysis of secure storage systems has received relatively little attention. (For instance, the literature seems to contain many more proofs about key-exchange protocols than about cryptographic file systems.)

More specifically, we study a pi calculus with file-system constructs. The calculus supports both access-control checks and a form of static scoping that limits the knowledge of terms—including file names and contents—to groups of clients. We design a system with secrecy types for the calculus. In this system, any type can be associated with a group of clients, which we call the reach of the type. By typing, we can then statically check certain secrecy properties, for instance, that a term is not leaked beyond the reach of its declared type. While the typing is static, it applies to a program subject to dynamic access-control checks.

For example, suppose that a client C_1 creates a secret m that it does not intend to share with other clients; it then writes m to a publicly known file. Further, suppose that another client C_2 attempts to read this file. We can analyze such a system in our calculus—this particular system typechecks only if C_2 does not have read access to that file. Indeed, if C_2 could read the file, m would no longer be secret to C_1. On the other hand, if C_2 does not attempt to read the file, it is possible that the system typechecks even if C_2 has read access. Various examples indicate that our type system is fairly permissive. Conversely, a soundness theorem states that any process that compromises secrecy intentions fails to typecheck. Further, typing has other interesting consequences; we derive, for instance, certain integrity properties.

Somewhat similar type systems exist for other calculi, including several pi calculi (see Section 7). The main novelty of our work is the investigation of file-system constructs, including access-control checks. This investigation requires new concepts and technical elements. It also enables us to treat examples that appear to be outside the scope of previous systems. The resulting secrecy properties, on the other hand, are fortunately standard.

In our calculus we can express and analyze programs that can request basic file operations and set file-operation permissions. We hide the details of file-system implemen-

*Also at Microsoft Research, Silicon Valley.

tations. Our intent is that many of those details should be addressed via translations (from high-level constructs to lower-level mechanisms) with security-preservation results (for instance, full abstraction results). We have taken some initial steps in this direction [8]. The present paper complements those steps, by providing a type discipline and proof principles that apply to a source language for those translations. Thus, the techniques developed in this paper can serve for establishing high-level secrecy guarantees, and those guarantees should carry over to lower-level systems obtained by translation.

The next section describes a file-system environment and some examples. Section 3 presents a pi calculus with file-system constructs and the system of secrecy types that we design for this calculus; Section 4 shows an operational semantics for the calculus. Section 5 defines a notion of secrecy, states our main results, and studies some consequences of the typing method. Section 6 enriches the file-system environment with directory structure and finer access control, and recovers the results of Section 5 with minor modifications to the calculus and the type system developed in Sections 3 and 4. Section 7 describes the context for our research, mentioning some related work. Section 8 concludes with a discussion of contributions and future work.

2 A file-system environment

We consider a distributed environment with a set of clients that interact among themselves and with a common file system. The file system stores data and maintains an access-control policy that is enforced on clients at the interface. This section describes that environment, introduces secrecy groups, and previews some examples, semi-formally. Later sections contain the relevant formal details.

2.1 The file-system and its clients

We assume that clients are indexed by a set $\mathcal{K}$, so that the set of clients is $\{C_k \mid k \in \mathcal{K}\}$. These indices also serve as subjects and administrators of access control. More concretely, there is a set of channels $\{\beta_k \mid k \in \mathcal{K}\}$ on which clients may send requests to the file system. Upon receiving a request on β_k, the file system decides whether the request is allowed by the access-control policy for the index k. (In an implementation, β_k may be realized with the use of an authenticated encryption key for client k.)

The file system organizes files in flat directories; we consider nested directories in Section 6. We assume that a number of directories and files are already known to the clients, and more files can be created under directories. Access-control rules come in three flavors: rules of the first kind give access rights to specific files; rules of the second kind give default access rights to arbitrary files under specific directories; rules of the third kind give rights to assert rules of the other two kinds.

In what follows, we distinguish between file *paths* and file *names*. A file name is atomic, as is a directory name. A file path (for example, $\mathsf{file}(d/f)$) is completely defined by a directory name (d) and a file name (f). Conversely, directory and file names may be used in different ways to construct different paths (so $\mathsf{file}(d/f)$ and $\mathsf{file}(d'/f)$ are distinct if $d \neq d'$). We sometimes call a file path simply a "file".

A *storage state* is a map from files paths to file contents. We focus on two operations on file contents, read and write, and an operation grant on file permissions. Yet another operation on file permissions, revoke, requires a more complicated treatment that we omit in this version of the paper.

Let $o \in \{\mathrm{W}, \mathrm{R}\}$ range over file operations (W for write, R for read). An *access-control policy* $\mathcal{R}$ is a set of rules of the form $o_k(d/*)$, $o_k(d/f)$, $\surd_{k'}(o_k(d/*))$, or $\surd_{k'}(o_k(d/f))$. For instance, a rule $o_k(d/f)$ would mean that client k is allowed to do operation o on file path $\mathsf{file}(d/f)$; a rule $\surd_{k'}(o_k(d/f))$ would mean that client k' is allowed to grant that permission. Rules $o_k(d/*)$ and $\surd_{k'}(o_k(d/*))$ have analogous meanings for default permissions on directory d.

2.2 Secrecy groups

We refer to certain subsets of client indices as groups. Some of those sets are induced by an access-control policy—for instance, the set of clients who have read access to a certain file (see Section 5.5). It is not true, however, that only those clients who have read access to a file may come to know its contents: a client who has access may read the contents, then share it with another client who is not allowed to read the file. While such sharing is often desirable, it is reasonable to try to limit its scope—we would want to know, for instance, if clients who have been granted access to sensitive files are leaking their contents, either intentionally or by mistake, to dishonest ones.

We use groups as a declarative means of specifying boundaries within which secrets may be shared. To make the definition of these groups more concrete, we draw a distinction between honest clients and potentially dishonest ones. Honest clients are those who play by the rules—they are disciplined in the way they interact with other clients and the file system, and this conformance may be checked statically by inspecting their code (*viz.* by typechecking). We let the set $\mathcal{I} \subseteq \mathcal{K}$ index honest clients. The remaining clients are assumed to be dishonest; in general they may make up an unknown, arbitrary attacker. *Secrecy groups* span either only subsets of honest clients (thus excluding all dishonest clients) or all clients (the group "public").

A *secrecy intention* is declared by stating that a certain name belongs to some group. In our type system, this dec-

laration is made by assuming a type for a name. In turn, a type can be associated with a secrecy group, called its *reach*. Informally, the reach of a type is the group within which the inhabitants of that type may be shared. Typing guarantees that secrecy intentions are never violated, *i.e.*, a name is never leaked outside the reach of its declared type.

2.3 Some examples (preview)

In what follows, we use a fairly standard pi-calculus syntax for writing code; the syntax of the calculus is presented formally in Section 3. We begin with the example sketched in Section 1. We return to this example and the others, giving additional details, in Section 4.

Let $\mathcal{K} = \{1, 2, 3\}$ be the set of client indices. Suppose that the code for the clients is $C_1 \mid C_2 \mid C_3$, with

$$\begin{aligned} C_1 &= (\nu m)\, \overline{\beta_1}\langle \mathsf{write}\, m, \mathsf{file}(d/f)\rangle \\ C_2 &= \overline{\beta_2}\langle \mathsf{read}\, p, \mathsf{file}(d/f)\rangle.\, p(x) \end{aligned}$$

and C_3 arbitrary code without β_1 or β_2 free. Thus C_1 invents a name m and requests that the file system write m to $\mathsf{file}(d/f)$; C_2 requests a read on the same file. The channel p is passed as a "return channel" (or continuation) to receive file contents. For a particular read request, the file system either sends back the contents of the file on the continuation (if the request succeeds) or does nothing (if it fails, *i.e.*, if the access-control policy does not allow the operation).

Suppose that we want to verify the following secrecy intentions. The name m should be known only within the group $\{1\}$. The term d is the name of a public directory intended to contain files with public names. The term f is a public file name; the contents of the file, however, should remain within $\{1\}$.

Let $\mathcal{R} = \{\mathrm{W}_1(d/f), \mathrm{R}_2(d/f)\}$. Thus clients C_1 and C_2 are allowed to write and read $\mathsf{file}(d/f)$, respectively; client C_3 does not have access to $\mathsf{file}(d/f)$. Clearly, however, these permissions are inconsistent with the secrecy intentions, since m can be leaked to C_2 via $\mathsf{file}(d/f)$. As we see in Section 3.5, this code does not typecheck. One might imagine obvious fixes such as preventing C_2 from reading $\mathsf{file}(d/f)$ by access control, or relaxing the scope of m to $\{1, 2\}$. These fixes make the code typecheck. (In the latter case, it is also necessary that p not be free in C_3.)

Next we discuss some more examples in order to demonstrate other features of our type system.

1. Let C_2 and C_3 be as above, but let

$$\begin{aligned} C_1 &= \overline{\beta_1}\langle \mathsf{grant}_2^{\mathrm{R}}, \mathsf{file}(d/f)\rangle. \\ &\qquad (\nu m)\, \overline{\beta_1}\langle \mathsf{write}\, m, \mathsf{file}(d/f)\rangle \end{aligned}$$

   and $\mathcal{R} = \{\mathrm{W}_1(d/f), \surd_1(\mathrm{R}_2(d/f))\}$. This definition allows client C_1 to grant client C_2 permission to read $\mathsf{file}(d/f)$, thus violating secrecy intentions, as above. A similar situation arises when $\mathcal{R} = \{\mathrm{W}_1(d/f), \surd_3(\mathrm{R}_2(d/f))\}$. Even though C_1's grant request fails, it is still possible that C_3 grants permission to C_2 to read $\mathsf{file}(d/f)$. In both cases, this code does not typecheck.

2. Now suppose that there is a fourth client C_4, so that the code for the clients is $C_1 \mid C_2 \mid C_3 \mid C_4$, and let

$$\begin{aligned} C_1 &= (\nu m)\, \overline{\beta_1}\langle \mathsf{write}\, m, \mathsf{file}(d/f)\rangle \\ C_2 &= p(x).\, \overline{\beta_2}\langle \mathsf{write}\, x, \mathsf{file}(d/f')\rangle \\ C_3 &= \overline{\beta_3}\langle \mathsf{read}\, p, \mathsf{file}(d/f)\rangle \\ C_4 &= \overline{\beta_4}\langle \mathsf{read}\, q, \mathsf{file}(d/f')\rangle.\, q(x) \end{aligned}$$

   and $\mathcal{R} = \{\mathrm{W}_1(d/f), \mathrm{R}_2(d/f), \mathrm{W}_2(d/f'), \mathrm{R}_4(d/f')\}$. This code typechecks, although a couple of observations make the code seem dangerous at first glance. First, C_2 has all the permissions that it had above, and more. Second, C_2 listens on the continuation passed by C_3 for reading the contents of $\mathsf{file}(d/f)$, and shares the received contents with C_4 via another file. It thus appears that C_3 is either collaborating with or using as "deputies" C_2 and C_4 to leak m. Secrecy intentions are still respected, however, because C_2 and C_4 are well-behaved by themselves (in particular, C_2 does not read from $\mathsf{file}(d/f)$), and C_3 can never read m from $\mathsf{file}(d/f)$ because access control would prevent it. So secrecy intentions are not violated (*i.e.*, m is never leaked to C_3, C_2, or C_4).

3. Now consider a dual scenario, with the two clients:

$$\begin{aligned} C_1 &= (\nu m)\, \overline{\beta_1}\langle \mathsf{read}\, n, \mathsf{file}(d/f)\rangle.\, n(x).\, \overline{x}\langle m\rangle \\ C_2 &= \overline{\beta_2}\langle \mathsf{write}\, p, \mathsf{file}(d/f)\rangle.\, p(x) \end{aligned}$$

   and $\mathcal{R} = \{\mathrm{R}_1(d/f), \mathrm{W}_2(d/f)\}$. This code does not typecheck. Consider the following run: C_2 writes the channel p on $\mathsf{file}(d/f)$; C_1 reads this channel from $\mathsf{file}(d/f)$, perhaps believing that it belongs to $\{1\}$, and communicates m on it. Since C_2 is listening on p, secrecy intentions are violated.

4. In this variant of the preceding example, default permissions come into play. Suppose that

$$\begin{aligned} C_1 &= (\nu m)\, (\nu f)\, \overline{q}\langle f\rangle. \\ &\qquad \overline{\beta_1}\langle \mathsf{read}\, n, \mathsf{file}(d/f)\rangle.\, n(x).\, \overline{x}\langle m\rangle \\ C_2 &= q(y).\, \overline{\beta_2}\langle \mathsf{write}\, p, \mathsf{file}(d/y)\rangle.\, p(x) \end{aligned}$$

   and $\mathcal{R} = \{\mathrm{R}_1(d/*), \mathrm{W}_2(d/*)\}$. Again, the code does not typecheck. Observe that the name f is not free in the code; it is generated during execution. However, both clients C_1 and C_2 enjoy default rights on d. Therefore, the same attack as in (3) may be played out after generating and sharing f.

5. Finally, consider the code $C_1 \mid C_2 \mid C_3$, where

$$\begin{aligned} C_1 &= (\nu m)\,(\nu f)\,\overline{q}\langle f\rangle. \\ &\quad\ \overline{\beta_1}\langle \mathsf{read}\ n, \mathsf{file}(d/f)\rangle.\ n(x).\ \overline{x}\langle m\rangle \\ C_2 &= q(y).\ \overline{\beta_2}\langle \mathsf{write}\ y, \mathsf{file}(d/f')\rangle \end{aligned}$$

and C_3 is arbitrary code without β_1, β_2, or n free, with $\mathcal{R} = \{\mathtt{R}_1(d/*), \mathtt{W}_2(d/*), \mathtt{R}_3(d/f')\}$. This code typechecks, although at first glance it seems as dangerous as the preceding one—C_1 shares the sensitive file name f with C_2, while C_2 has default write permissions on all files in d, including f; to make matters worse, C_2 writes f to a public file that C_3 can read. Secrecy intentions are still respected, however: C_2 does not use f to access the file system, and neither can C_3 because access control would prevent it (*cf.* (2), that typechecks in much the same manner). One obvious way to violate secrecy intentions would be to give C_3 default write permissions on d, in which case the code would of course fail to typecheck.

3 A typed pi calculus with file constructs

We use a polyadic synchronous pi calculus for writing and verifying client code. This section gives the syntax of terms, types, and processes, then presents our type system for this calculus, and finally considers examples.

3.1 Terms, types, and typed processes

Terms are defined by the following grammar, where $k \in \mathcal{K}$ and $o \in \{\mathtt{W}, \mathtt{R}\}$.

$M, N ::=$	terms
$m, n, p, q, d, f, \ldots, \beta_k$	name, request channel
$\mathsf{write}\ M$	write term to file
$\mathsf{read}\ M$	read file on channel
grant_k^o	grant permissions
$\mathsf{file}(M/N)$	file path
$x, y, \ldots$	variable

We use $u, v, \ldots$ to represent names or variables. The terms $\mathsf{write}\ M$ and $\mathsf{read}\ M$ are file-operation commands that respectively mean "write contents M to" and "read on channel M contents from" a file. The term grant_k^o means "grant client k permission to do o on" a directory or file.

We assume a set of clients indexed by $\mathcal{K}$, and a subset of honest clients indexed by $\mathcal{I} \subseteq \mathcal{K}$. We let $\mathcal{G}$ range over subsets of $\mathcal{I}$. We do not consider proper subsets of $\mathcal{K}$ that contain indices in $\mathcal{K} - \mathcal{I}$ since, intuitively, a dishonest client may share a term that it knows with any of the other clients. The notation $\widetilde{\varphi}$ stands for a (possibly empty) vector $\varphi_1, \ldots, \varphi_n$. Types are defined by the following grammar:

$\mathcal{H} ::=$	groups
$\mathcal{G}$	trusted group
$\mathcal{K}$	untrusted group
$T^\nu ::=$	declared types
$\mathcal{G}[\widetilde{T}]$	polyadic channel
Un	untrusted
$\mathcal{H}\{T\}$	file name
$T ::=$	types
T^ν	declared type
$\mathcal{H}'/\mathcal{H}$	directory name
$\mathsf{Wr}(T)$	write contents
$\mathsf{Rd}(T)$	read contents
Gr_k	grant permissions
$\mathsf{Req}_i\ (i \in \mathcal{I})$	request channel
$\#\mathcal{H}'/\mathcal{H}\{T\}$	file path

Declared types are types with which new names can be declared in the calculus (according to the syntax of typed processes, below). We introduce a type sort $\mathcal{H}\{T\}$ for declaring file names, and the sorts $\mathcal{H}'/\mathcal{H}$ and $\#\mathcal{H}'/\mathcal{H}\{T\}$ for typing directory names and file paths respectively.

- The type $\mathcal{H}\{T\}$ is given to a file name; this typing means that the file name may be shared within the group $\mathcal{H}$, while the contents may be shared within the reach of the type T. This type construct is somewhat similar in form to the traditional type construct $\mathcal{G}[\widetilde{T}]$ for symmetric channels [7]. However, the name of a symmetric channel can never be less secret than its contents, since by knowing its name one can know what it carries. On the other hand, a file with a publicly known name may contain secrets, and these contents may be protected by dynamic access control.

- The type $\mathcal{H}'/\mathcal{H}$ is given to a directory name; this typing means that the directory name may be shared within the group $\mathcal{H}'$, and may contain files whose names may be shared within the group $\mathcal{H}$. Note that we do not let clients declare new directories in this particular system; since default permissions are based on directory names, files created under new directories would not inherit any interesting access control. (We remove this simplification in Section 6.)

- The type $\#\mathcal{H}'/\mathcal{H}\{T\}$ is given to a file path $\mathsf{file}(d/f)$; this typing means that the directory name d has type $\mathcal{H}'/\mathcal{H}$, and the file name f has type $\mathcal{H}\{T\}$. The file path may be shared within the group $\mathcal{H}' \cap \mathcal{H}$, while the contents may be shared within the reach of the type T.

- The types $\mathsf{Wr}(T)$ and $\mathsf{Rd}(T)$ are respectively given to commands for writing and reading file contents of type T.

- The type Gr_k is given to a term of the form grant_k^o.

- The type Req_i is given to an honest client's request channel β_i.
- The special type Un is given to untrusted terms, typically those that an attacker may know.

For a type T, the group within which the inhabitants of that type may be shared is given by its reach $\|T\|$, defined next:

$$\begin{array}{rclcrcl} \|\mathcal{G}[\widetilde{T}]\| & = & \bigcap \|\widetilde{T}\| \cap \mathcal{G} & & \|\mathsf{Wr}(T)\| & = & \|T\| \\ \|\mathsf{Un}\| & = & \mathcal{K} & & \|\mathsf{Rd}(T)\| & = & \|T\| \\ \|\mathcal{H}'/\mathcal{H}\| & = & \mathcal{H}' & & \|\mathsf{Gr}_k\| & = & \mathcal{K} \\ \|\mathcal{H}\{T\}\| & = & \mathcal{H} & & \|\mathsf{Req}_i\| & = & \{i\} \\ \|\#\mathcal{H}'/\mathcal{H}\{T\}\| & = & \mathcal{H}' \cap \mathcal{H} & & & & \end{array}$$

Any type whose reach is $\mathcal{K}$ is called a *public* type; a term that belongs to $\mathcal{K}$ is a public term, and is said to have "unrestricted scope". We sometimes use the phrase "scope of a term" to mean the reach of its type, when this type is clear from context.

Note that the reach of a file-name type $\mathcal{H}\{T\}$ or a file-path type $\#\mathcal{H}'/\mathcal{H}\{T\}$ does not depend on the content type T. Note also that a directory name may be public even if it is intended to contain files with secret names or contents. Finally, the definition of reach also reflects that an honest client never shares its request channel with any other client; this restriction is relaxed somewhat in Section 5.4.

The grammar for processes is fairly standard:

$P, Q ::=$	processes
$\overline{u}\langle\widetilde{M}\rangle.P$	output
$u(\widetilde{x}).P$	input
0	nil
$P \mid Q$	composition
$!P$	replication
$(\nu n : T^{\nu})P$	restriction

A departure from the usual polyadic pi calculus is in the inclusion of a declared type T^{ν} in a restriction. Intuitively, this type indicates a secrecy intention. Notice that the language of typed processes depends on the set $\mathcal{I}$. However, type declarations do not affect runtime behaviour, and the same "untyped" process can be type-annotated in several different ways to verify various secrecy intentions.

The notions of free names and variables (fn and fv) and closed expressions are as usual; so are various abbreviations (*e.g.*, Π for indexed parallel compositions).

3.2 Preliminaries on types

Typechecking a system involves typechecking the clients and the access-control policy under the same assumptions. For clients, the typechecking applies to honest clients; these clients are restricted in their use of channels and file requests, by the typing rules shown in Section 3.3. For the access-control policy, the typechecking imposes restrictions on the set of file operations that the policy may grant to dishonest clients; these restrictions are specified in Section 3.4.

The partition between honest and dishonest clients plays a central role in typechecking the system. The code for the clients as well as the access-control policy impose typing constraints that finally determine whether the partition is valid, *i.e.*, whether all honest clients are well-typed, and whether the access-control policy is suitably restrictive for the remaining (possibly dishonest) ones. Arriving at the correct partition may be delicate: overestimating the set of honest clients does not help if one of those clients is ill-typed; underestimating this set imposes more constraints on the access-control policy. Once we do have a valid partition, however, we can prove that an honest client (or indeed a subset of honest clients) can protect secrets from all other (honest and dishonest) clients.

We define typing relations $\vdash_{\mathcal{L}}$ for sets of indices $\mathcal{L}$. For now, the reader may assume that $\mathcal{L}$ ranges over singleton sets $\{k\}$ (where $k \in \mathcal{K}$) and the set $\mathcal{K}$. However, the typing rules apply for every $\mathcal{L} \subseteq \mathcal{K}$ (see Section 5.4). Informally, typing under $\vdash_{\mathcal{L}}$ requires $\mathcal{L}$ to intersect the reach of every type given by the relation. An honest client of index $i \in \mathcal{I}$ is typechecked under the relation $\vdash_{\{i\}}$. The client may know terms that belong to any group that includes i; however it may not know any term that belongs to a group that does not include i. On the other hand, $\vdash_{\mathcal{K}}$ is the "most liberal" typing relation, since $\mathcal{K}$ trivially intersects all groups. For instance, the file system is typechecked under $\vdash_{\mathcal{K}}$, since its data structures are available to all clients. More general uses of $\vdash_{\mathcal{L}}$ appear in Section 5.4, where we typecheck the code of clients indexed by $\mathcal{L}$ in combination.

3.3 Typing judgments and typing rules

Type environments contain typing assumptions for channels, file names, directory names, and variables, and are defined by the following grammar.

$\Gamma ::=$	environments
$\varnothing$	empty
$u : T, \Gamma$	type declaration

If Γ contains a declaration $u : T$, we say that $u \in \mathsf{dom}(\Gamma)$ and $\Gamma(u) = T$. There are three kinds of typing judgments: $\Gamma \vdash \diamond$ (good environment), $\Gamma \vdash_{\mathcal{L}} M : T$ (well-typed term), and $\Gamma \vdash_{\mathcal{L}} P$ (well-typed process). The typing rules are shown in Figure 1. In these rules, whenever the consequent is of the form $\Gamma \vdash_{\mathcal{L}} M : T$ or $\Gamma \vdash_{\mathcal{L}} P$, and the antecedent has no judgments for well-typed terms or well-typed processes, the judgment $\Gamma \vdash \diamond$ is implicitly included in the antecedent. We also let $i, i' \in \mathcal{I}$, $j \in \mathcal{K} - \mathcal{I}$, and $k \in \mathcal{K}$. Some of the rules are discussed next.

The rule (Term u) checks that the reach of a type T intersects the typing group $\mathcal{L}$. This rule is similar to those for

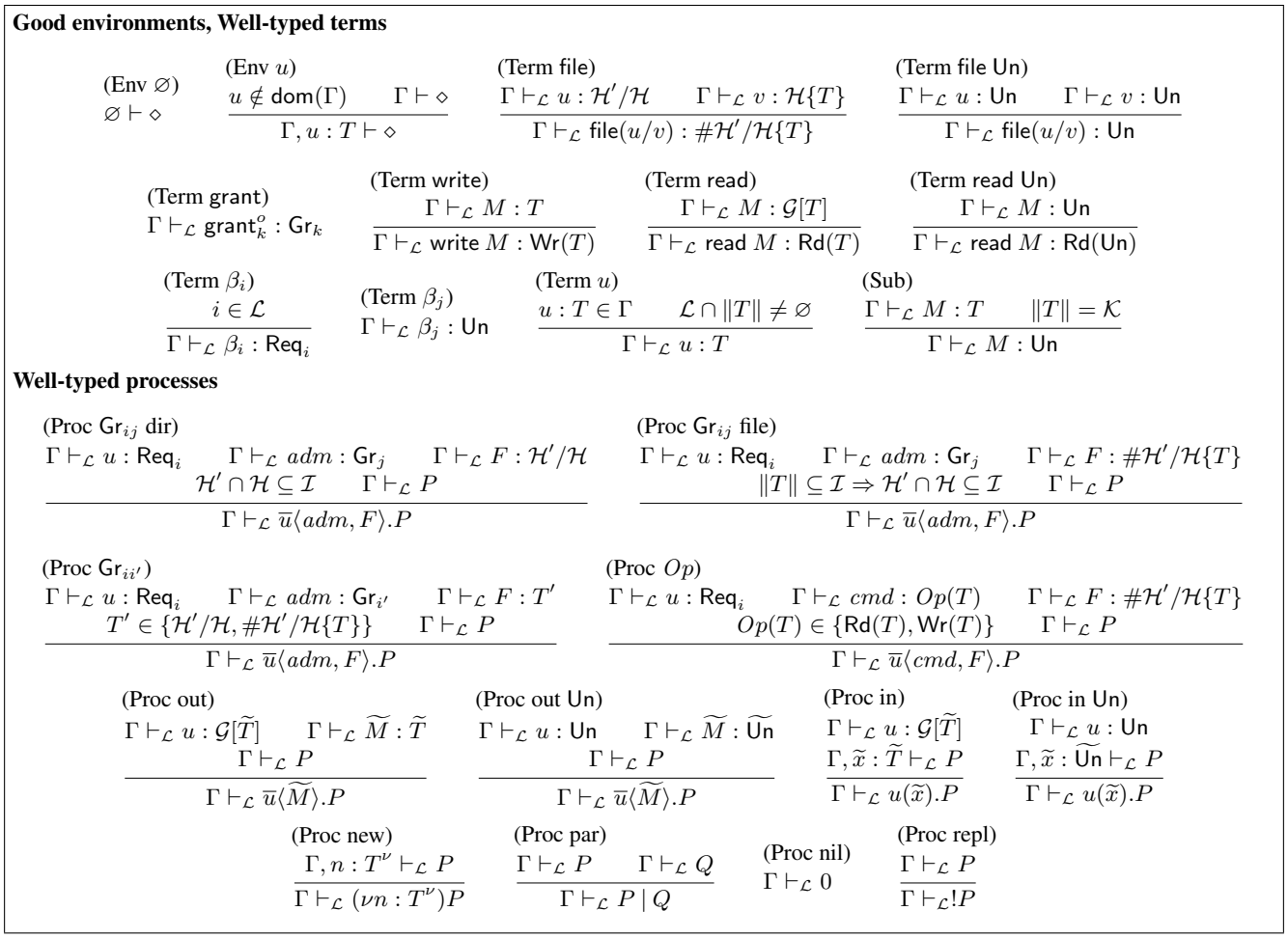

Figure 1. Typing rules for environments, terms, and processes

judging good types in group creation [7], for example. The rule (Term β_i) assigns the type Req_i to β_i if i indexes an honest client in $\mathcal{L}$; the rule (Term β_j) assigns the untrusted type to β_j for all j that index dishonest clients. Request channels with indices in $\mathcal{I} - \mathcal{L}$ cannot be assigned types because of scoping constraints. The rule (Proc Op) forces honest clients to be in the reach of the type of file contents if they request operations from the file system; recall that, in general, file paths may be in scope even if their contents are not. The rules (Proc Gr_{ij} dir) and (Proc Gr_{ij} file) check that honest clients can never grant dishonest clients permission to access public files with trusted contents. (The rules do not prevent dishonest clients from obtaining permission to access files whose paths they can never know.) On the other hand, (Proc $\mathsf{Gr}_{ii'}$) allows an honest client to be able to grant arbitrary permissions to other honest clients, since honest clients would never misuse permissions. The rule (Sub) is a subtyping rule—it allows any public type to be treated as untrusted. As a result, public directories and public file names may be used by and shared between all clients. Of course, (Sub) is not invertible; in particular, an honest client may not receive a public file name on an untrusted channel and then request a file operation on it. Some typical "Trojan horse" attacks are thus ruled out.

It is easy to see that the relation $\vdash$ is monotonic over typing groups, *i.e.*, if $\Gamma \vdash_{\mathcal{L}} P$ then $\Gamma \vdash_{\mathcal{L}'} P$ for any $\mathcal{L}' \supseteq \mathcal{L}$. The following proposition says that a client with public free names and no secrecy intentions can always be typed. This result is similar to ones that allow typing of untyped processes in related type systems (*e.g.*, [7]).

Proposition 3.1. *We say that a process is intention-free if all declared types in it have reach $\mathcal{K}$. For any closed intention-free client C_k, if $\|\Gamma(n)\| = \mathcal{K}$ for all $n \in \mathsf{fn}(C_k)$, then $\Gamma \vdash_{\{k\}} C_k$.*

3.4 Type constraints on file-system states

File-system states are treated as special processes. A file-system state is typed under the same assumptions that type

the system of clients, following (Proc par). Further, since the file system is exposed to all clients, we type it under the most liberal typing group, $\mathcal{K}$. Let ζ range over file-system states. A state ζ has three components: $\zeta.\mathcal{R}$, which is a set of access-control rules (the access-control policy); $\zeta.\rho$, which is a function that maps file names to file contents (the storage state); and $\zeta.\gamma$, which is a function that associates channel names with buffered queues, where each buffer contains terms pending to be returned on the channel. Intuitively, each channel in the domain of $\zeta.\gamma$ is a continuation passed by possibly many read requests, and the buffer for that channel contains the contents to be sent back in the order the requests were received by the file system.

In the following, we write $\vdash$ to mean $\vdash_{\mathcal{K}}$. We introduce three more kinds of judgments: $\Gamma \vdash \zeta.\mathcal{R}$ (good access-control policy), $\Gamma \vdash \zeta.\rho$ (good storage state), and $\Gamma \vdash \zeta.\gamma$ (good return heap).

Access-control policy $\Gamma \vdash \zeta.\mathcal{R}$ iff $\mathsf{fn}(\zeta.\mathcal{R}) \subseteq \mathsf{dom}(\Gamma)$ and for all $j, j' \in \mathcal{K} - \mathcal{I}$, $o \in \{\mathrm{W}, \mathrm{R}\}$, and names f and d,

1. if either $o_j(d/*) \in \zeta.\mathcal{R}$ or $\surd_{j'}(o_j(d/*)) \in \zeta.\mathcal{R}$, then it is never the case that $\Gamma \vdash d : \mathcal{K}/\mathcal{K}$; and
2. if $\Gamma \vdash \mathsf{file}(d/f) : \#\mathcal{K}/\mathcal{K}\{T\}$ for some T, and either $o_j(d/f) \in \zeta.\mathcal{R}$ or $\surd_{j'}(o_j(d/f)) \in \zeta.\mathcal{R}$, then $\|T\| = \mathcal{K}$.

Storage state $\Gamma \vdash \zeta.\rho$ iff $\mathsf{fn}(\zeta.\rho) \subseteq \mathsf{dom}(\Gamma)$ and for all files $F \in \mathsf{dom}(\zeta.\rho)$, either of the following holds:

- $\Gamma \vdash \zeta.\rho(F) : T$ and $\Gamma \vdash F : \#\mathcal{H}'/\mathcal{H}\{T\}$ for some $\mathcal{H}'$, $\mathcal{H}$, and T.
- $\Gamma \vdash \zeta.\rho(F) : \mathsf{Un}$ and if $\Gamma \vdash F : \#\mathcal{H}'/\mathcal{H}\{T\}$ for some $\mathcal{H}'$, $\mathcal{H}$, and T, then $\|T\| = \mathcal{K}$.

Return heap $\Gamma \vdash \zeta.\gamma$ iff $\mathsf{fn}(\zeta.\gamma) \subseteq \mathsf{dom}(\Gamma)$ and for all channels $n \in \mathsf{dom}(\zeta.\gamma)$ and all terms M in $\zeta.\gamma(n)$, either of the following holds:

- $\Gamma \vdash M : T$ and $\Gamma \vdash n : \mathcal{G}[T]$ for some $\mathcal{G}$ and T.
- $\Gamma \vdash M : \mathsf{Un}$ and $\Gamma \vdash n : \mathsf{Un}$.

The typing constraints on good access-control policies allow dishonest clients to access only those public files whose contents are public. Accordingly, dishonest clients may not have default permissions on public directories with public files. Further, dishonest clients may not grant themselves any of these potentially dangerous permissions. These conditions are similar in spirit to the rules (Proc Gr_{ij} dir), (Proc Gr_{ij} file), and (Proc $\mathsf{Gr}_{ii'}$) for processes. A storage state is well-typed if for every file in its domain, the type of its contents matches the type of its path. A return heap is well-typed if for every channel in the domain of the heap, the type of terms in its buffer matches the type of its name. A file-system state ζ is compatible with Γ, written $\Gamma \vdash \zeta$, iff $\Gamma \vdash \zeta.\mathcal{R}$, $\Gamma \vdash \zeta.\rho$, and $\Gamma \vdash \zeta.\gamma$.

3.5 The examples, revisited

Recall the initial example of Section 2.3. We reflect its assumptions by letting the type declaration for m in C_1 be $\mathcal{G}_1[\,]$, and Γ be partially specified as follows, using the abbreviations $\mathcal{G}_1 = \{1\}$ and $\mathcal{G}_{12} = \{1, 2\}$:

$$\Gamma = f : \mathcal{K}\{\mathcal{G}_1[\,]\}, d : \mathcal{K}/\mathcal{K}, p :?$$

Suppose that we stipulate that $\mathcal{I} = \{1, 2\}$, that is, C_1 and C_2 are honest but C_3 may not be. Under this partition, $\Gamma \vdash \mathcal{R}$ (since $\mathcal{R}$ does not give any rights to C_3). However, even while $\Gamma \vdash_{\{1\}} C_1$, we have $\Gamma \nvdash_{\{2\}} C_2$ for all type assumptions $\Gamma(p)$. Indeed, by (Proc Op), $\Gamma(p)$ must be of the form $\mathcal{G}[\mathcal{G}_1[\,]]$, whose reach being contained in $\{1\}$ cannot intersect $\{2\}$. Thus C_2 cannot be honest, and a similar argument rules out honesty for C_3 as well—therefore $\mathcal{I}$ is forced to be just $\{1\}$.

Note that C_3 can be typed under $\vdash_{\{3\}}$, even if it tries to access $\mathsf{file}(d/f)$, since the file's type can be subsumed by Un. In general a dishonest client can always be typed under an environment which types each free name with unrestricted scope (*i.e.*, by subsumption, with the untrusted type). Similarly, we may assume C_2 to be dishonest, and by letting $\Gamma(p) = \mathsf{Un}$, we may have $\Gamma \vdash_{\{2\}} C_2$. However, then $\Gamma \nvdash \mathcal{R}$, since $\mathcal{R}$ gives rights to C_2 on a file with trusted contents. Thus we conclude that $\mathcal{R}$ is not consistent with the declared intentions.

When we reduce $\mathcal{R}$ to $\{\mathrm{W}_1(d/f)\}$, typing becomes consistent. Conversely, revising secrecy assumptions in Γ also works—for example, we may allow m's scope to be extended to $\{1, 2\}$ by declaring its type as $\mathcal{G}_{12}[\,]$, reflecting the changes in a new environment, say

$$\Gamma = f : \mathcal{K}\{\mathcal{G}_{12}[\,]\}, d : \mathcal{K}/\mathcal{K}, p : \mathcal{G}_{12}[\mathcal{G}_{12}[\,]]$$

Now, with the original assumption of $\mathcal{I} = \{1, 2\}$, C_1 and C_2 may be checked as honest, and $\Gamma \vdash \mathcal{R}$.

We briefly address the other examples next.

1. (Does not typecheck.) As above, C_2 and C_3 cannot be honest. C_1 cannot be honest either, since $\Gamma \nvdash_{\{1\}} C_1$—the rule (Proc Gr_{ij} file) does not apply, since $\Gamma \vdash \mathsf{file}(d/f) : \#\mathcal{K}/\mathcal{K}\{\mathcal{G}_1[\,]\}$. With C_1 dishonest, $\Gamma \nvdash \mathcal{R}$.

2. (Typechecks.) Let $\mathcal{I} = \{1, 2, 4\}$ and

   $$\Gamma = f : \mathcal{K}\{\mathcal{G}_1[]\}, d : \mathcal{K}/\mathcal{K}, f' : \mathcal{K}\{\mathsf{Un}\}, p : \mathsf{Un}, q : \mathsf{Un}$$

3. (Does not typecheck.) We begin with the partially specified assumptions

   $$\Gamma = f : \mathcal{K}\{?\}, d : \mathcal{K}/\mathcal{K}, n :?, p :?$$

   Suppose that $\Gamma(f) = \mathcal{K}\{\mathsf{Un}\}$. Then C_1 cannot be honest, since neither Un nor any $\mathcal{G}[\mathsf{Un}]$ as $\Gamma(n)$ allows a

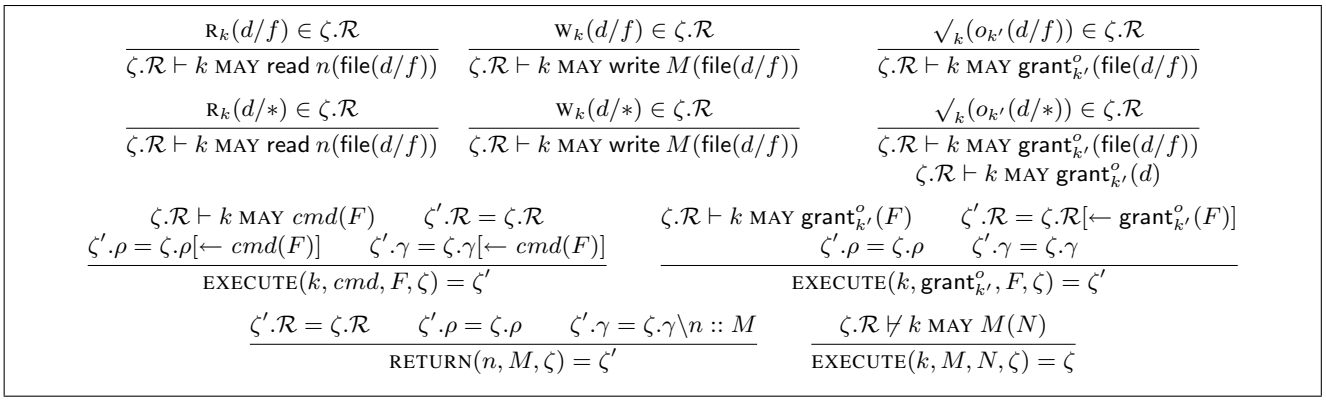

Figure 2. Semantics of file-system functions

consistent type assumption for x. If, on the other hand, we assume C_1 to be honest, then $\Gamma(f) = \mathcal{K}\{\mathcal{G}[\mathcal{G}_1[\,]]\}$ and $\Gamma(n) = \mathcal{G}'[\mathcal{G}[\mathcal{G}_1[\,]]]$ for some $\mathcal{G}, \mathcal{G}'$. Then C_2 cannot be honest, since by (Proc Op), $\Gamma(p) = \mathcal{G}[\mathcal{G}_1[\,]]$ whose reach being contained in $\subseteq \{1\}$ cannot intersect $\{2\}$. Since at least one of C_1 and C_2 is dishonest, $\Gamma \not\vdash \mathcal{R}$.

4. (Does not typecheck.) We begin with the partially specified assumptions

$$\Gamma = q :?, d : \mathcal{K}/\mathcal{K}, n :?, p :?$$

If $\Gamma(q) = \mathsf{Un}$ then C_2 cannot be honest, so $\Gamma \not\vdash \mathcal{R}$. Suppose that $\Gamma(q) = \mathcal{G}[T]$ for some $\mathcal{G}$, where T is the type declaration of f in C_1. Much as in (3), at least one of C_1 and C_2 is dishonest, so $\Gamma \not\vdash \mathcal{R}$.

5. (Typechecks.) Let $\mathcal{I} = \{1, 2\}$, let the type declaration of the new f in C_1 be $\mathcal{K}\{\mathcal{G}_1[\mathcal{G}_1[\,]]\}$, and let

$$\Gamma = q : \mathsf{Un}, f' : \mathcal{K}\{\mathsf{Un}\}, d : \mathcal{K}/\mathcal{K}, n : \mathcal{G}'[\mathcal{G}[\mathcal{G}_1[\,]]]$$

for any $\mathcal{G}', \mathcal{G} \subseteq \{1, 2\}$.

4 Operational semantics for the calculus

The operational semantics for the calculus is built on a standard commitment relation for closed processes (see, *e.g.*, [1]). File-system states are treated as special processes; a commitment relation on such states is described next.

A file-system state ζ has components $\zeta.\mathcal{R}$, $\zeta.\rho$, and $\zeta.\gamma$ as described in Section 3.4. Access-control judgments are of the form $\zeta.\mathcal{R} \vdash \cdot \text{ MAY } \cdot$; they are defined in Figure 2. There, cmd ranges over write M and read M.

Let $\widehat{\mathsf{file}(d/f)} = d/f$, and $\widehat{d} = d/*$. Let μ range over file-operation permissions, of the form $o_k(d/f)$ or $o_k(d/*)$. With $\zeta.\mathcal{R}\{\mu\}$, the permissions μ are added to $\zeta.\mathcal{R}$. With $\zeta.\rho\{F \mapsto M\}$, $\zeta.\rho$ is augmented with the new mapping $F \mapsto M$. With $\zeta.\gamma\{n :: M\}$, M is queued to the buffer of n in $\zeta.\gamma$, and with $\zeta.\gamma\backslash n :: M$, M is dequeued from the buffer of n in $\zeta.\gamma$. Let

$$\begin{aligned}
\zeta.\mathcal{R}[\leftarrow \mathsf{grant}^o_k(F)] &= \zeta.\mathcal{R}\{o_k(\widehat{F})\} \\
\zeta.\rho[\leftarrow \mathsf{read}\ n(F)] &= \zeta.\rho \\
\zeta.\rho[\leftarrow \mathsf{write}\ M(F)] &= \zeta.\rho\{F \mapsto M\} \\
\zeta.\gamma[\leftarrow \mathsf{read}\ n(F)] &= \zeta.\gamma\{n :: \zeta.\rho(F)\} \\
\zeta.\gamma[\leftarrow \mathsf{write}\ M(F)] &= \zeta.\gamma
\end{aligned}$$

The functions EXECUTE and RETURN modify file-system states, as shown in Figure 2. Then we have the following commitment rules for file-system states.

$$\frac{\forall M, N : \zeta'\{M/x, N/y\} = \text{EXECUTE}(k, M, N, \zeta)}{\zeta \xrightarrow{\beta_k} (x, y).\ \zeta'}$$

$$\frac{\zeta' = \text{RETURN}(n, M, \zeta)}{\zeta \xrightarrow{\overline{n}} \langle M \rangle.\ \zeta'}$$

Both rules represent a step from ζ to ζ'. In the former, we have the communication of input values for (x, y) on β_k. In the latter, we have an output M on n.

Additional rules complete the definition of the commitment relation, in particular defining silent steps $P \xrightarrow{\tau} P'$; those rules are standard.

5 Properties of well-typed systems

This section presents our main results for the type system, namely subject reduction and secrecy. It also explores some related topics: integrity guarantees, treatment of client collusions, and a characterization of file-access groups.

5.1 Type preservation

The principal property of a well-typed system is that each part of the system remains well-typed during sys-

tem execution. More concretely, if several processes and a file-system state are typed under various respective typing groups using the same type environment, then they remain well-typed under the respective groups after an arbitrary number of reductions of their parallel composition.

Proposition 5.1 (Subject reduction)**.** *Let* $\Gamma \vdash_{\mathcal{L}_\theta} P_\theta$ *for each* $\theta \in \Theta$*,* $\Gamma \vdash \zeta$*, and* $\Pi_{\theta\in\Theta} P_\theta \mid \zeta \xrightarrow{\tau}{}^{\star} (\nu \widetilde{n} : \widetilde{T})\, (\Pi_{\theta\in\Theta} P'_\theta \mid \zeta')$*. Then* $\Gamma, \widetilde{n} : \widetilde{T} \vdash_{\mathcal{L}_\theta} P'_\theta$ *and* $\Gamma, \widetilde{n} : \widetilde{T} \vdash \zeta'$*.*

Subject reduction has a number of consequences; the most important is a secrecy theorem for well-typed systems, which we discuss next.

5.2 Secrecy by typing and access control

We view an attacker as arbitrary code that interacts with the system via dishonest clients. An attacker is modeled by its knowledge, which is a set of names, and is an upper bound on the set of free names in its code (see [2, 7] for similar analyses). Let S range over such sets of names.

Definition 5.2 (S-adversary)**.** *A closed process* E *is an* S*-adversary if* E *is intention-free (*i.e.*, all declared types in it have reach* $\mathcal{K}$*) and* $\mathsf{fn}(E) \subseteq S$*.*

Next, we provide a definition of secrecy, using the usual notion of escape (similar to that in, *e.g.*, [2, 7]). We slightly generalize the adversary, by letting it contain a well-behaved part (some process Q), and an arbitrary part (some S-adversary). A term is revealed if it may eventually be published on a channel known to the adversary. A term is a $(\cdot \triangleright \mathcal{G})$-secret if its type suggests that it should not be leaked outside the group $\mathcal{G}$.

Definition 5.3 (Secrecy)**.** *Let* P *and* Q *be closed processes,* S *be a set of names,* ζ *be a file-system state, and* M *be a closed term. Let* $\widetilde{m} = \mathsf{fn}(M) - \mathsf{fn}(P, Q, S, \zeta)$*.*

1. *Under the assumptions* $\widetilde{m} : \widetilde{T}$*,* P *reveals* M *to* (Q, S) *via* ζ *if* $P \mid Q \mid E \mid \zeta \xrightarrow{\tau}{}^{\star} \xrightarrow{\overline{c}} (\nu \widetilde{m} : \widetilde{T})\, \langle \widetilde{M} \rangle$*.* R *for some* S*-adversary* E*,* $c \in \mathsf{fn}(Q) \cup S$*,* $\widetilde{M} \ni M$*, and process* R*.*
2. *If* $\Gamma \vdash M : T$ *with* $\|T\| \subseteq \mathcal{G}$*, then* M *is a* $(\Gamma \triangleright \mathcal{G})$*-secret.*

We now define well-typed systems and state the main secrecy theorem. Let $\mathbb{C}_\mathcal{L}$ denote the process $\Pi_{k\in\mathcal{L}} C_k$.

Definition 5.4 (Well-typed system)**.** *A system* $(\mathbb{C}_\mathcal{I}, \zeta)$ *is well-typed under hypotheses* Γ *if* $\Gamma \vdash_{\{i\}} C_i$ *for each* $i \in \mathcal{I}$*, and* $\Gamma \vdash \zeta$*.*

In other words, a system is well-typed if, using some common type assumptions, it is possible to type each honest client under its singleton group, and the file-system state under $\mathcal{K}$. The remaining clients can be trivially typed if they do not know any trusted names initially and are intention-free. Subject reduction yields the following theorem.

Theorem 5.5 (Secrecy by typing and access control)**.** *Suppose that* $(\mathbb{C}_\mathcal{I}, \zeta)$ *is well-typed under* Γ*. Further suppose that* $\|\Gamma(s)\| = \mathcal{K}$ *for each* $s \in S$*, where* $\mathcal{K} \supset \mathcal{I}$*. Then for all groups* $\mathcal{G} \subseteq \mathcal{I}$ *and fresh assumptions* Γ'*,* $\mathbb{C}_\mathcal{G}$ *does not reveal any* $(\Gamma, \Gamma' \triangleright \mathcal{G})$*-secret to* $(\mathbb{C}_{\mathcal{I}-\mathcal{G}}, S)$ *via* ζ*.*

Thus in a well-typed system, any secret meant to be shared only within a subset of honest clients is never revealed to the other (honest and dishonest) clients.

As a special case, let $\mathcal{I}$ be the singleton set $\{1\}$, so that $\mathcal{G} = \mathcal{I}$ is the only non-empty trusted group. Suppose that $\Gamma \vdash_{\{1\}} P$, $\Gamma \vdash \zeta$, and for all $s \in \mathsf{dom}(\Gamma)$, $\|\Gamma(s)\| = \mathcal{K}$, where $\mathcal{K} \supset \{1\}$. Let $n : \{1\}[\,]$ be a new name declared inside P. Then P does not reveal n to $(0, \mathsf{dom}(\Gamma))$ via ζ.

5.3 Integrity consequences

While above we focus on secrecy properties, the type system also yields integrity properties. Such properties can be specified by declaring "expectations", and verified statically with our type system. More concretely, an expectation specifies that certain terms should have certain types, but has no observable effect. The types in question have both structure (to distinguish, for example, channels from files) as well as reach (in particular, to reason about sources of messages). For an honest client with index $i \in \mathcal{I}$, let

$$\text{expect}(\widetilde{M} : \widetilde{T}).\ P \stackrel{\text{def}}{=} (\nu n : T_e)\, (\overline{n}\langle \widetilde{M} \rangle \mid n(\widetilde{x}).\ P)$$

where $n \notin \mathsf{fn}(P)$, $x \notin \mathsf{fv}(P)$, and $T_e = \{i\}[\widetilde{T}]$. Such expectations can be verified statically, by typechecking, although the terms $\widetilde{M}$ may, of course, contain variables instantiated at runtime.

5.4 Reasoning under client collusions

When talking about well-typed systems in Section 5.2, we require honest clients to be typed under their respective singleton groups. In this section we relax the conditions on well-typedness of systems by letting honest clients "collude", that is, be typed under a combined group that contains each of their respective indices. Well-typedness of processes is robust under arbitrary collusions with dishonest clients: for all non-empty $\mathcal{L} \subseteq \mathcal{K}$, $\Gamma \vdash_\mathcal{L} P$ if and only if $\Gamma \vdash_{\mathcal{L}\cup\mathcal{J}} P$, where $\mathcal{J} \subseteq \mathcal{K} - \mathcal{I}$. (Basically, this property holds because the typing requirements are vacuous for clients with indices outside $\mathcal{I}$.) Therefore, when reasoning under collusions that involve both honest and dishonest

clients, it is sufficient to put only the honest-client indices in the typing groups.

Collusions may arise when a group of honest clients who share a secret want to protect the secret from the rest of the clients. These remaining clients are then assumed to act adversarially by colluding. Specifically, when reasoning about $(\cdot \triangleright \mathcal{G})$-secrets, we allow clients indexed by $\mathcal{G}$, and those indexed by $\mathcal{I} - \mathcal{G}$, to form a pair of collusions and typecheck accordingly. In particular, this means that clients in $\mathcal{G}$ may share their request channels, and those in $\mathcal{I} - \mathcal{G}$ can do the same. In fact, it is not uncommon to have clients share access capabilities in file-system environments. As the following theorem shows, our type system can support reasoning about such situations.

Theorem 5.6 (Secrecy by collective typing and access control)**.** *Suppose that* $\Gamma \vdash_{\mathcal{G}} \mathbb{C}_{\mathcal{G}}$*,* $\Gamma \vdash_{\mathcal{I}-\mathcal{G}} \mathbb{C}_{\mathcal{I}-\mathcal{G}}$*, and* $\Gamma \vdash \zeta$*. Further suppose that* $\|\Gamma(s)\| = \mathcal{K}$ *for each* $s \in S$*, where* $\mathcal{K} \supset \mathcal{I}$*. Then for all fresh assumptions* Γ'*,* $\mathbb{C}_{\mathcal{G}}$ *does not reveal any* $(\Gamma, \Gamma' \triangleright \mathcal{G})$*-secret to* $(\mathbb{C}_{\mathcal{I}-\mathcal{G}}, S)$ *via* ζ*.*

5.5 Characterizing file-access groups in well-typed systems

Given some code, we say that the set of clients who can eventually do a particular operation o on a file F is the *o-access group* for F. In this section, we conservatively characterize file-access groups for well-typed code, by analyzing type constraints and permission constraints in combination. For some file type $\#\mathcal{H}'/\mathcal{H}\{T\}$ and state ζ, let

$$\begin{aligned}
\textsc{Have}^o_k(d/f) &= \{o_k(d/f), o_k(d/*)\} \\
\textsc{Get}^o_k(d/f) &= \{\surd_{k'}(o_k(d/f)) \text{ s.t. } k' \in \mathcal{H}' \cap \mathcal{H}\} \cup \\
&\quad \{\surd_{k'}(o_k(d/*)) \text{ s.t. } k' \in \mathcal{H}'\} \\
\textsc{Access}^o(d/f) &= \{k \in \mathcal{H}' \cap \mathcal{H} \cap \|T\| \text{ s.t.} \\
&\quad \zeta.\mathcal{R} \cap (\textsc{Have}^o_k(d/f) \cup \textsc{Get}^o_k(d/f)) \neq \varnothing\}
\end{aligned}$$

Intuitively, the function ACCESS calculates upper bounds on the sets of clients that, starting from a given state, can eventually execute particular operations on files of a given type. This set is determined by permissions as well as typing restrictions. (For example, honest clients must be in the scope of the file name and its contents to request an operation on a file.) The calculation uses two auxiliary functions HAVE and GET: HAVE collects existing permissions for access to a file, and GET collects permissions that may allow administrators to grant access to a file eventually. We use typing restrictions to refine the latter set as well, since administrators need to be (at least partially) in the scope of the file they grant access to.

Definition 5.7. *Suppose that* $\Gamma \vdash_{\mathcal{I}} \mathbb{C}_{\mathcal{I}}$*,* $\Gamma \vdash \zeta$*, and* E *is a closed intention-free process such that* $\|\Gamma(s)\| = \mathcal{K}$ *for each* $s \in \mathsf{fn}(E)$*. Further suppose that* $\mathbb{C}_{\mathcal{I}} | E | \zeta \xrightarrow{\tau}{}^{\star} (\nu\Gamma')(\mathbb{P} | \zeta')$*, and* $\mathbb{P} \xrightarrow{\overline{\beta_k}} (\nu\Gamma'') \langle cmd, F \rangle . \mathbb{Q}$*, so that* $\Gamma, \Gamma', \Gamma'' \vdash F : T_F$*. Then we say that well-typed code at* ζ *can eventually have* k *sending* cmd *on* $F : T_F$ *at* ζ'*.*

The following theorem states that the calculation of ACCESS is sound (or conservative), that is, $\textsc{Access}^o(d/f)$ actually computes an upper bound on the o-access group for $\mathsf{file}(d/f)$.

Theorem 5.8 (Access-group analysis)**.** *Let* $\widehat{\mathsf{read}}\ \cdot = \text{R}$ *and* $\widehat{\mathsf{write}}\ \cdot = \text{W}$*. Let* ACCESS *be as defined for file type* $\#\mathcal{H}'/\mathcal{H}\{T\}$ *and file-system state* ζ*, and suppose that well-typed code at* ζ *can eventually have* k *sending* cmd *on* $F : \#\mathcal{H}'/\mathcal{H}\{T\}$ *at* ζ'*. If* $\zeta'.\mathcal{R} \vdash k \text{ MAY } cmd(F)$*, then* $k \in \textsc{Access}^{\widehat{cmd}}(\widehat{F})$*.*

6 Finer access control with new directories

The granularity of default permissions in an access-control policy is always restricted by the structure of directories at that point. We may hope to enrich access-control administration by letting clients create and declare new directories. However, an access-control policy cannot specify non-trivial permissions for new directories in a flat directory structure. We therefore adapt our calculus to allow hierarchical (nested) directories.

We define a file path by a *directory path* and a file name, where a directory path is a non-empty string of directory names. We let clients declare new directory names by extending the grammar of declared types. We extend the sort of terms with directory paths, and the sort of types with directory-path types.

$M ::=$	terms
$\cdots$	others as above
$\mathsf{dir}(\widetilde{M})$	directory path
$\mathsf{file}(\widetilde{M}/N)$	file path
$T^\nu ::=$	declared types
$\cdots$	others as above
$\mathcal{H}'/\mathcal{H}$	directory name
$T ::=$	types
$\cdots$	others as above
$\#\widetilde{\mathcal{H}}'/\mathcal{H}$	directory path type
$\#\widetilde{\mathcal{H}}'/\mathcal{H}\{T\}$	file path type

A directory path $\mathsf{dir}(d_1 \ldots d_n)$ has type $\#\mathcal{H}_1 \ldots \mathcal{H}_n/\mathcal{H}_{n+1}$ if for each $k \in 1 \ldots n$, the name d_k has type $\mathcal{H}_k/\mathcal{H}_{k+1}$. Reaches for the new type sorts are defined by:

$$\|\#\widetilde{\mathcal{H}}'/\mathcal{H}\| = \bigcap \widetilde{\mathcal{H}}' \qquad \|\#\widetilde{\mathcal{H}}'/\mathcal{H}\{T\}\| = \bigcap \widetilde{\mathcal{H}}' \cap \mathcal{H}$$

Access-control policies have rules of the form $o_k(\widetilde{d}/f)$, $o_k(\widetilde{d}/*)$, $\surd_{k'}(o_k(\widetilde{d}/f))$, or $\surd_{k'}(o_k(\widetilde{d}/*))$. The meanings of the first three sorts are analogous to those in Section 4.

Terms

$$\frac{\Gamma \vdash_{\mathcal{L}} u : \mathcal{H}'/\mathcal{H}}{\Gamma \vdash_{\mathcal{L}} \mathsf{dir}(u) : \#\mathcal{H}'/\mathcal{H}}\ \text{(Term dir)} \qquad \frac{\Gamma \vdash_{\mathcal{L}} \mathsf{dir}(\widetilde{u}) : \#\widetilde{\mathcal{H}}''/\mathcal{H}' \qquad \Gamma \vdash_{\mathcal{L}} v : \mathcal{H}'/\mathcal{H}}{\Gamma \vdash_{\mathcal{L}} \mathsf{dir}(\widetilde{u}v) : \#\widetilde{\mathcal{H}}''\mathcal{H}'/\mathcal{H}}\ \text{(Term dir)}$$

$$\frac{\Gamma \vdash_{\mathcal{L}} u_1 : \mathsf{Un} \quad \ldots \quad \Gamma \vdash_{\mathcal{L}} u_n : \mathsf{Un}}{\Gamma \vdash_{\mathcal{L}} \mathsf{dir}(u_1 \ldots u_n) : \mathsf{Un}}\ \text{(Term dir Un)}$$

$$\frac{\Gamma \vdash_{\mathcal{L}} \mathsf{dir}(\widetilde{u}) : \#\widetilde{\mathcal{H}}'/\mathcal{H} \qquad \Gamma \vdash_{\mathcal{L}} v : \mathcal{H}\{T\}}{\Gamma \vdash_{\mathcal{L}} \mathsf{file}(\widetilde{u}/v) : \#\widetilde{\mathcal{H}}'/\mathcal{H}\{T\}}\ \text{(Term file)} \qquad \frac{\Gamma \vdash_{\mathcal{L}} \mathsf{dir}(\widetilde{u}) : \mathsf{Un} \qquad \Gamma \vdash_{\mathcal{L}} v : \mathsf{Un}}{\Gamma \vdash_{\mathcal{L}} \mathsf{file}(\widetilde{u}/v) : \mathsf{Un}}\ \text{(Term file Un)}$$

Access-control policy $\Gamma \vdash \zeta.\mathcal{R}$ iff $\mathsf{fn}(\zeta.\mathcal{R}) \subseteq \mathsf{dom}(\Gamma)$ and for all $j, j' \in \mathcal{K} - \mathcal{I}$, $o \in \{\mathrm{W}, \mathrm{R}\}$, and names f and name strings $\widetilde{d}$,

1. if either $o_j(\widetilde{d}/*) \in \zeta.\mathcal{R}$ or $\surd_{j'}(o_j(\widetilde{d}/*)) \in \zeta.\mathcal{R}$, then it is never the case that $\Gamma \vdash \mathsf{dir}(\widetilde{d}) : \#\widetilde{\mathcal{K}}/\mathcal{K}$; and
2. if $\Gamma \vdash \mathsf{file}(\widetilde{d}/f) : \#\widetilde{\mathcal{K}}/\mathcal{K}\{T\}$ for some T, and either $o_j(\widetilde{d}/f) \in \zeta.\mathcal{R}$ or $\surd_{j'}(o_j(\widetilde{d}/f)) \in \zeta.\mathcal{R}$, then $\|T\| = \mathcal{K}$.

Figure 3. New typing rules for terms and policies

The last sort allows k' to grant k access permissions for all files and subdirectories that are recursively constructed under $\widetilde{d}$; in particular, $\surd_{k'}(\mathrm{R}_k(\widetilde{d}/*))$ would allow k' to grant the permissions $\mathrm{R}_k(\widetilde{d}/*)$, $\mathrm{R}_k(\widetilde{d}/f)$, $\mathrm{R}_k(\widetilde{d}d/*)$, $\mathrm{R}_k(\widetilde{d}d/f)$, and so on.

The remaining semantic rules of Section 4 apply by replacing d with $\widetilde{d}$ or $\mathsf{dir}(\widetilde{d})$ as appropriate. Figure 3 shows some new typing rules for terms and access-control policies. The remaining typing rules of Sections 3.3 and 3.4 apply by replacing $\mathcal{H}'$ with $\widetilde{\mathcal{H}}'$ or $\bigcap \widetilde{\mathcal{H}}'$ as appropriate. The function ACCESS of Section 5.5 can be similarly redefined.

Our theorems carry over as they are for this richer file-system environment. Although the proofs are technically no harder than those for the simplified environment studied in Sections 2–5, the interplay between access control and typing gets even more interesting.

For example, let $\mathcal{R} = \{\surd_1(\mathrm{R}_2(d/*)), \surd_3(\mathrm{R}_4(d/*))\}$, and suppose that C_2 wants to create a file somewhere under d and prevent C_4 from reading that file. Then it can create a new subdirectory d' under d, type d' so that C_3 is not in its scope, and safely create a new file f under d'. Since C_3 cannot know the name d', it cannot grant (default or plain) access to $\mathsf{file}(dd'/f)$ to C_4.

7 Related work

Our type system extends previous ones so as to deal with access-control checks. It is particularly close to an intermediate type system developed in the study of group creation [7]. It goes beyond that type system by introducing secrecy types for file-system constructs, in such a way that dynamic access-control checks, together with static scoping, play a role in guaranteeing secrecy of file contents.

Kirli's mobility regions [14] for distributed functional programs are similar to groups as presented here. Bugliesi *et al.* develop another calculus that uses group creation to specify discretionary access-control policies [6]; they include a sophisticated enforcement of "delivery rules" that control the flow of values. Ideas similar to group creation also appear in the work of Braghin *et al.* on a calculus for role-based access control [4]. It is however not immediate to see how to apply these approaches to our setting. In [6], for example, it is possible for clients to *specify* file-access groups declaratively; in our setting, file-access groups are influenced by external access control—we declare, instead, our intentions about file access, and verify that external access control respects such intentions.

As in most access-control systems, and as in the study of group creation, in this paper secrecy is not defined as the absence of certain flows of information (that is, as some sort of non-interference property). Rather, secrecy is presented as the impossibility of certain communication events (for instance, sending a message that contains a particular sensitive value). One may however imagine many possible variants, dealing with other concepts of secrecy, and also with authenticity properties beyond the ones verifiable in our system (*e.g.*, [12]). We leave the investigation of such variants for further work.

The recent literature also includes a few calculi with constructs for authorization. In particular, Fournet *et al.* develop a spi calculus with authorization assertions [10]; a type system for that calculus serves for checking generalized correspondence assertions (rather than secrecy properties, as in this paper).

Several other works emphasize distribution. Thus, in the language KLAIM [9], a type system checks that processes have been granted the necessary right to perform operations at specified localities [16]. Hennessy and Riely describe a typing system for a distributed pi-calculus that ensures that agents cannot access the resources of a system without first being granted the capability to do so [13]. Bugliesi *et al.* explore access-control types for the calculus of boxed ambients [5] with a typing relation similar in form to ours, but without dynamic access control—access control is specified

in terms of static security levels.

Yet another research direction addresses access control in languages such as Java. Thus, Banerjee and Naumann examine the use of access control for secure information flow in that setting [3]. Pottier *et al.* develop type systems that guarantee the success of access checks [17]. In contrast, our type system does not guarantee the success of access checks; indeed, type soundness depends on the failure of some of those checks.

8 Conclusion

In this paper we investigate the interplay of secrecy types with access-control checks in the setting of a fairly standard file system. As indicated in the introduction, our goal is to enable the analysis of programs that use the file system; the details of the file-system implementation can then be refined while preserving secrecy properties.

In addition to further research on the file-system implementation, there are a number of other opportunities for further work. These include the treatment of integrity properties, which sometimes follow from our secrecy properties but which may well deserve a first-class treatment. These also include relating file-system concepts and techniques to those explored in other settings. For instance, one might wonder about encodings back and forth between our calculus and various process calculi with group creation. We have started to consider those encodings; in particular, while it is not too hard to encode the original calculus with group creation into our calculus, it is not immediate to see how to go in the other direction. We anticipate that a primitive treatment of files, directories, and access-control policies is more likely to be fruitful as a basis for specifying and verifying file-system security properties.

Acknowledgments We thank Ricardo Corin for his helpful comments. This work was partly supported by the National Science Foundation under Grants CCR-0204162, CCR-0208800, and CCF-0524078, and by Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under Contract B554869.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387–415, 2003.

[3] A. Banerjee and D. Naumann. Using access control for secure information flow in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 155–169, 2003.

[4] C. Braghin, D. Gorla, and V. Sassone. A distributed calculus for role-based access control. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 48–60, 2004.

[5] M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: The calculus of boxed ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004.

[6] M. Bugliesi, D. Colazzo, and S. Crafa. Type based discretionary access control. In *International Conference on Concurrency Theory (CONCUR)*, pages 225–239. Springer LNCS, 2004.

[7] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.

[8] A. Chaudhuri and M. Abadi. Formal security analysis of basic network-attached storage. In *ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 43–52, 2005.

[9] R. de Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[10] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *European Symposium on Programming (ESOP)*, pages 141–156. Springer LNCS, 2005.

[11] H. Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, 1999.

[12] A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1-3):379–409, 2003.

[13] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *High-Level Concurrent Languages (HLCL)*, pages 3–17. Elsevier Science Publishers, 1998.

[14] Z. D. Kirli. Confined mobile functions. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 283–294, 2001.

[15] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. Reed. Strong security for network-attached storage. In *Conference on File and Storage Technologies (FAST)*, pages 1–13. USENIX Association, 2002.

[16] R. D. Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[17] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, 2005.